1/19

Hamdy Soliman

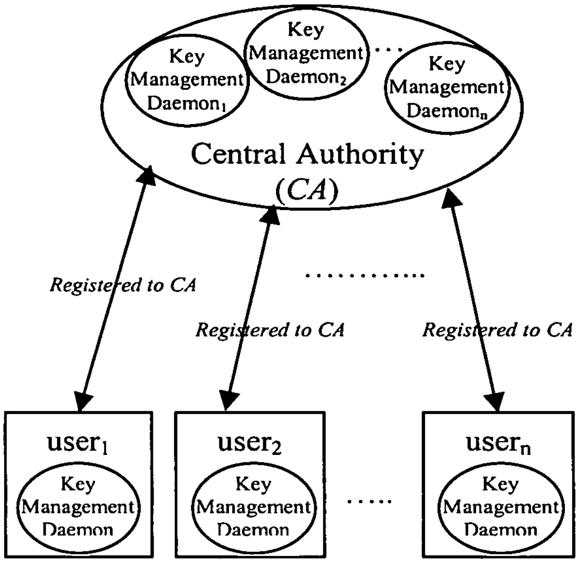
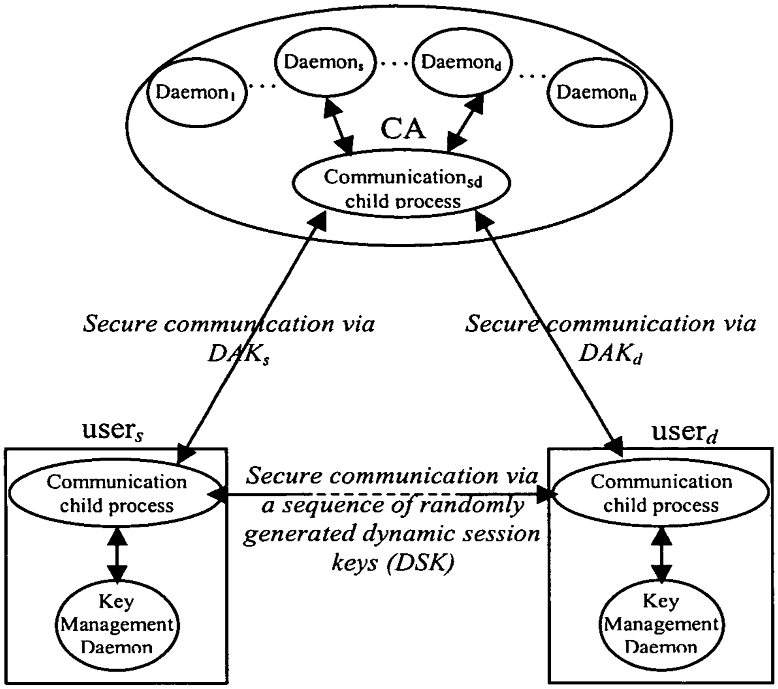


FIG. 1a



Daemon₁
Daemonₛ
Daemon_d
Daemonₙ
CA
Communication_sd child process
Secure communication via DAKₛ
Secure communication via DAK_d
userₛ
Communication child process
Key Management Daemon
Secure communication via a sequence of randomly generated dynamic session keys (DSK)
user_d
Communication child process
Key Management Daemon

FIG. 1b

Hamdy Soliman

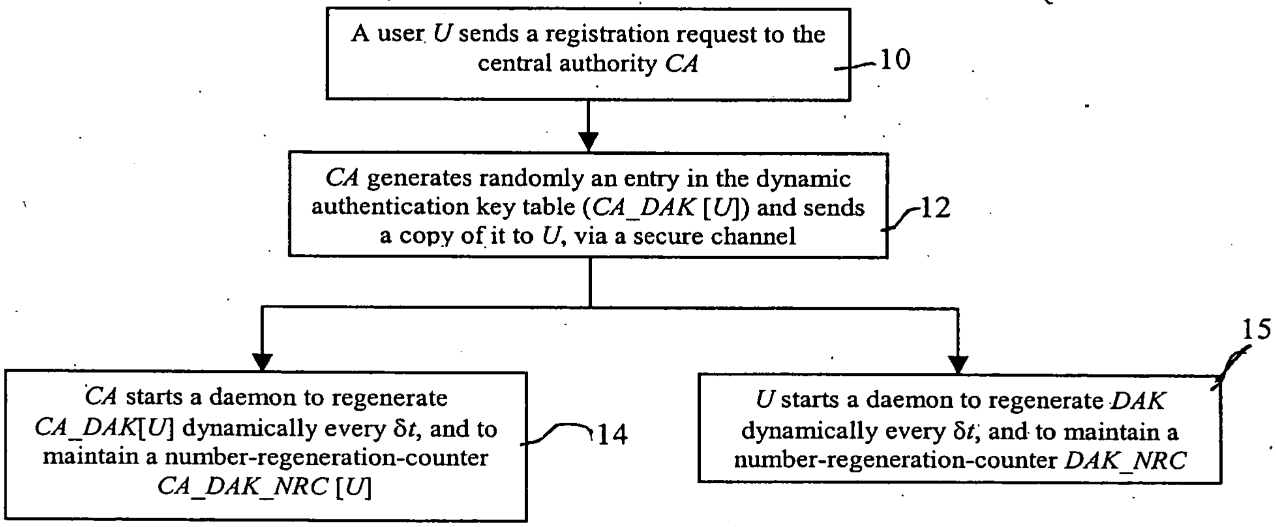

A user U sends a registration request to the central authority CA
10
CA generates randomly an entry in the dynamic authentication key table (CA_DAK [U]) and sends a copy of it to U, via a secure channel
12
CA starts a daemon to regenerate CA_DAK[U] dynamically every δt, and to maintain a number-regeneration-counter CA_DAK_NRC [U]
14
U starts a daemon to regenerate DAK dynamically every δt, and to maintain a number-regeneration-counter DAK_NRC
15

FIG. 2

Hamdy Soliman

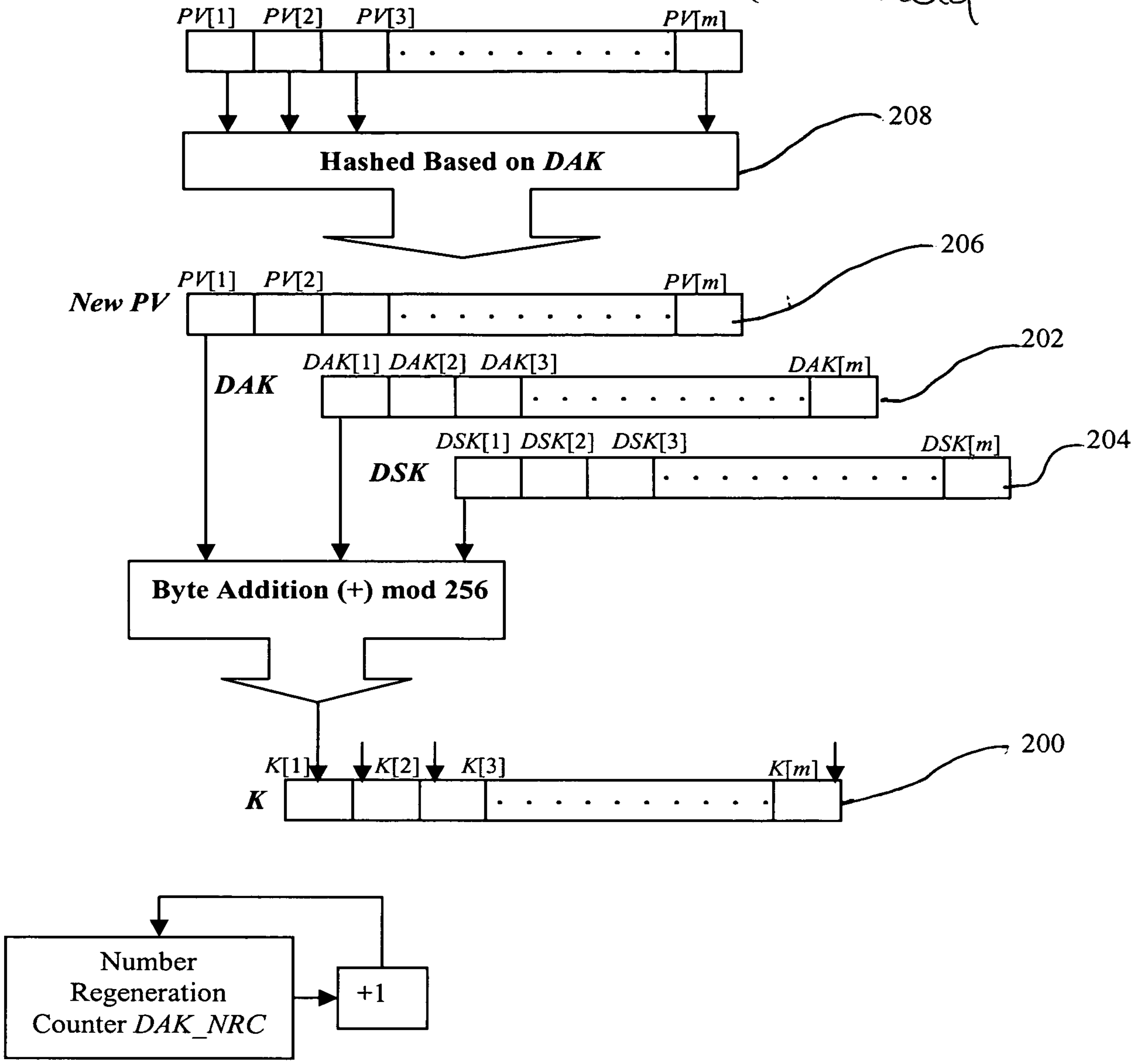


FIG. 3a

4/19
Hamdy Soliman

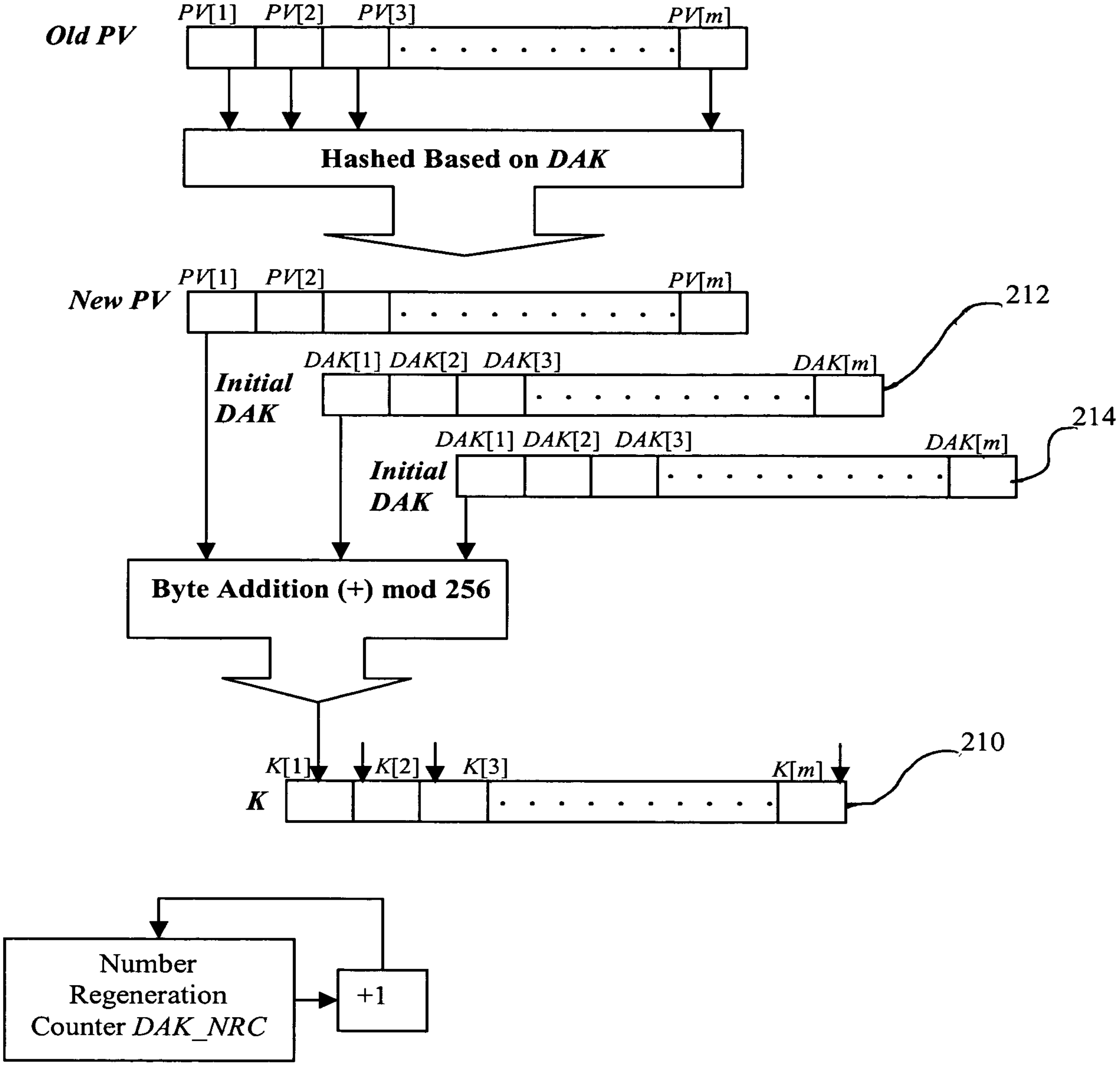


FIG. 3b

Old PV

5/19

Hamdy Soliman

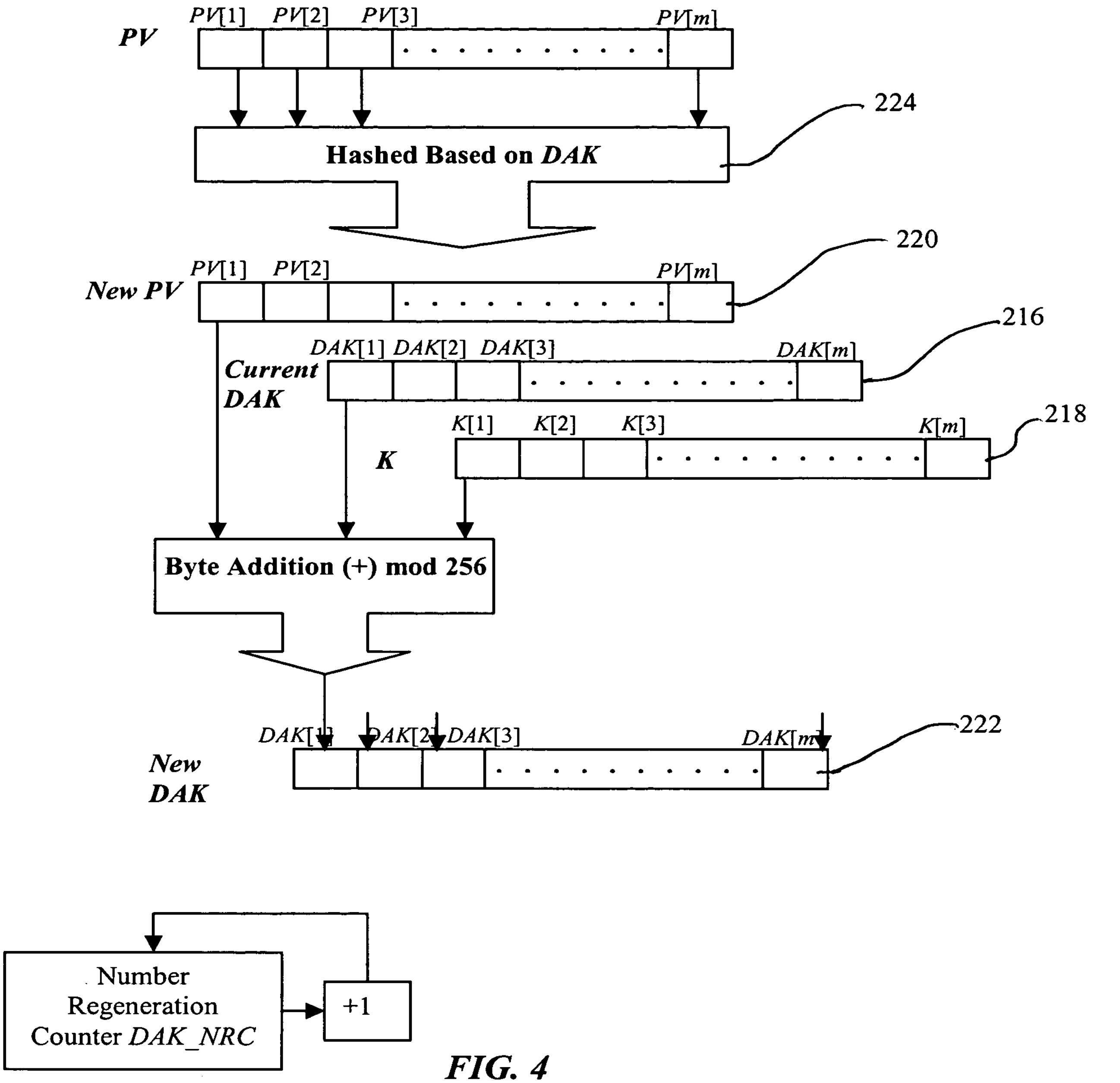


FIG. 4

6/19 Hamdy Soliman

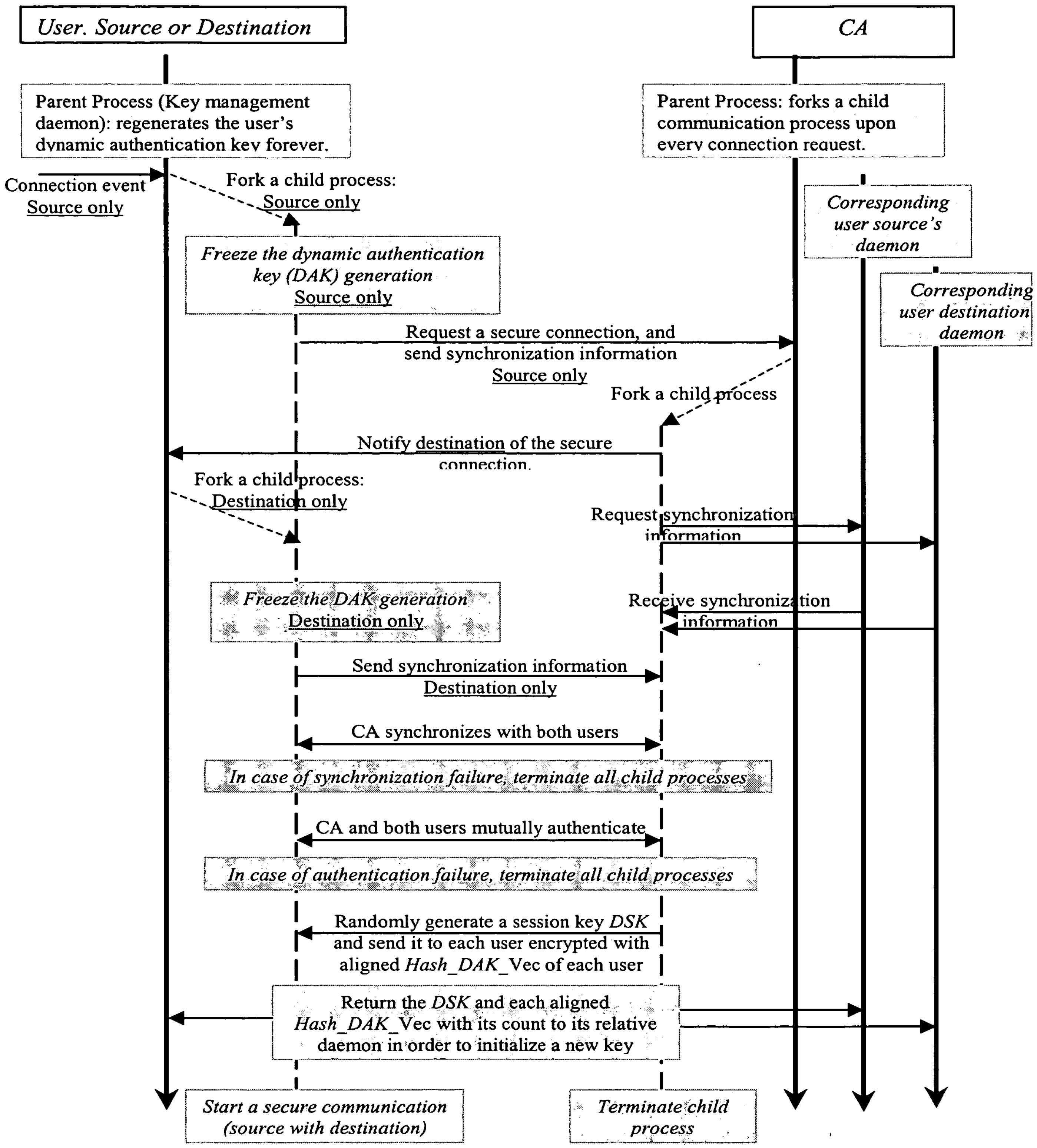

User. Source or Destination
CA
Parent Process (Key management daemon): regenerates the user's dynamic authentication key forever.
Parent Process: forks a child communication process upon every connection request.
Connection event
Source only
Fork a child process:
Source only
Corresponding user source's daemon
Freeze the dynamic authentication key (DAK) generation
Source only
Corresponding user destination daemon
Request a secure connection, and send synchronization information
Source only
Fork a child process
Notify destination of the secure connection.
Fork a child process:
Destination only
Request synchronization information
Freeze the DAK generation
Destination only
Receive synchronization information
Send synchronization information
Destination only
CA synchronizes with both users
In case of synchronization failure, terminate all child processes
CA and both users mutually authenticate
In case of authentication failure, terminate all child processes
Randomly generate a session key DSK and send it to each user encrypted with aligned Hash_DAK_Vec of each user
Return the DSK and each aligned Hash_DAK_Vec with its count to its relative daemon in order to initialize a new key
Start a secure communication (source with destination)
Terminate child process

FIG. 5

7/19 Hamdy Soliman

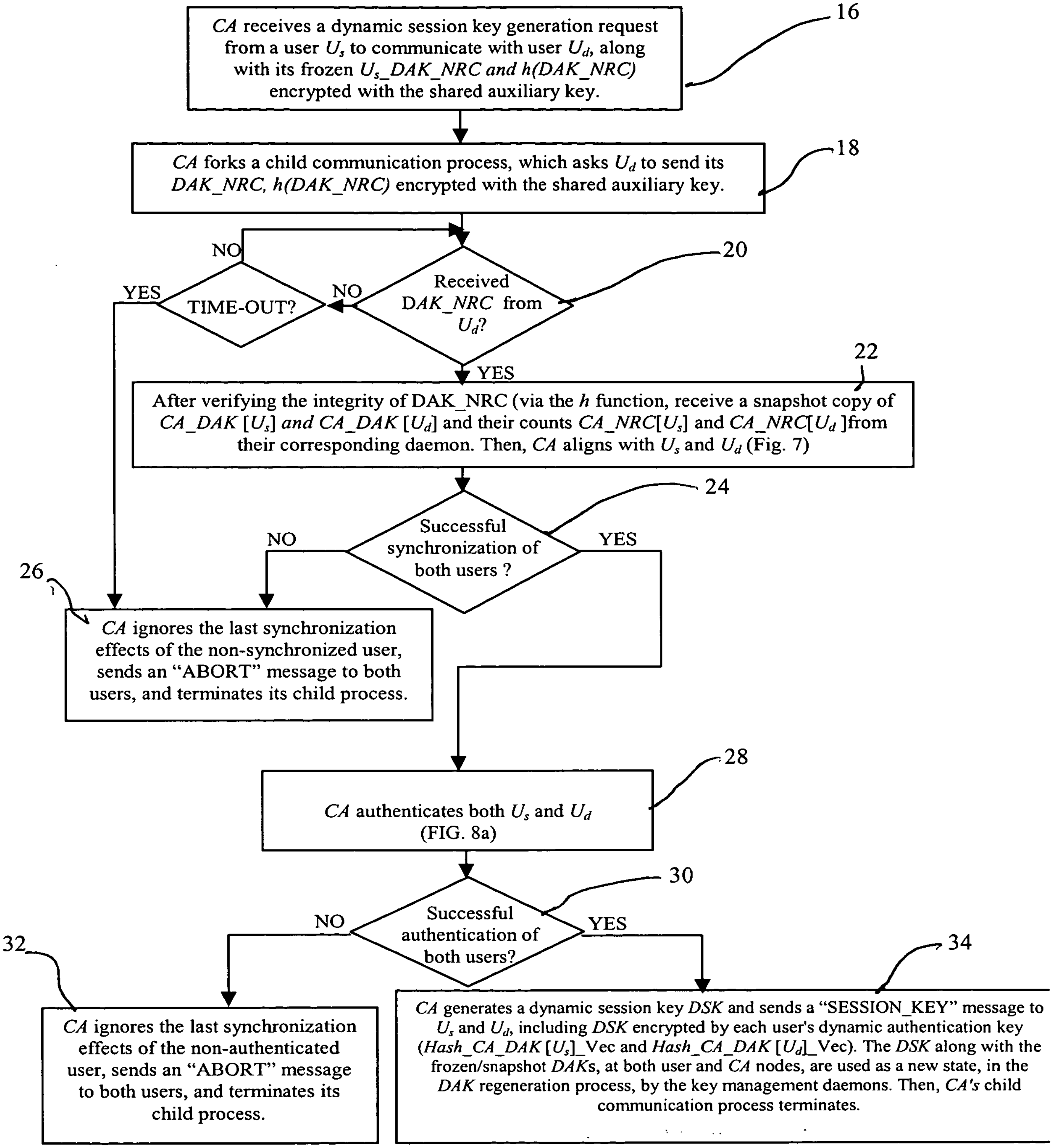

CA receives a dynamic session key generation request from a user U_s to communicate with user U_d, along with its frozen U_s*_DAK_NRC and h(DAK_NRC)* encrypted with the shared auxiliary key.
16
CA forks a child communication process, which asks U_d to send its *DAK_NRC, h(DAK_NRC)* encrypted with the shared auxiliary key.
18
NO
YES
TIME-OUT?
NO
Received D*AK_NRC* from U_d?
20
YES
22
After verifying the integrity of DAK_NRC (via the *h* function, receive a snapshot copy of *CA_DAK* [U_s] *and CA_DAK* [U_d] and their counts *CA_NRC*[U_s] and *CA_NRC*[U_d]from their corresponding daemon. Then, *CA* aligns with U_s and U_d (Fig. 7)
24
NO
Successful synchronization of both users ?
YES
26
CA ignores the last synchronization effects of the non-synchronized user, sends an "ABORT" message to both users, and terminates its child process.
28
CA authenticates both U_s and U_d (FIG. 8a)
30
NO
Successful authentication of both users?
YES
32
34
CA ignores the last synchronization effects of the non-authenticated user, sends an "ABORT" message to both users, and terminates its child process.
CA generates a dynamic session key *DSK* and sends a "SESSION_KEY" message to U_s and U_d, including *DSK* encrypted by each user's dynamic authentication key (*Hash_CA_DAK* [U_s]_Vec and *Hash_CA_DAK* [U_d]_Vec). The *DSK* along with the frozen/snapshot *DAK*s, at both user and *CA* nodes, are used as a new state, in the *DAK* regeneration process, by the key management daemons. Then, *CA's* child communication process terminates.

FIG. 6

Hamdy Soliman

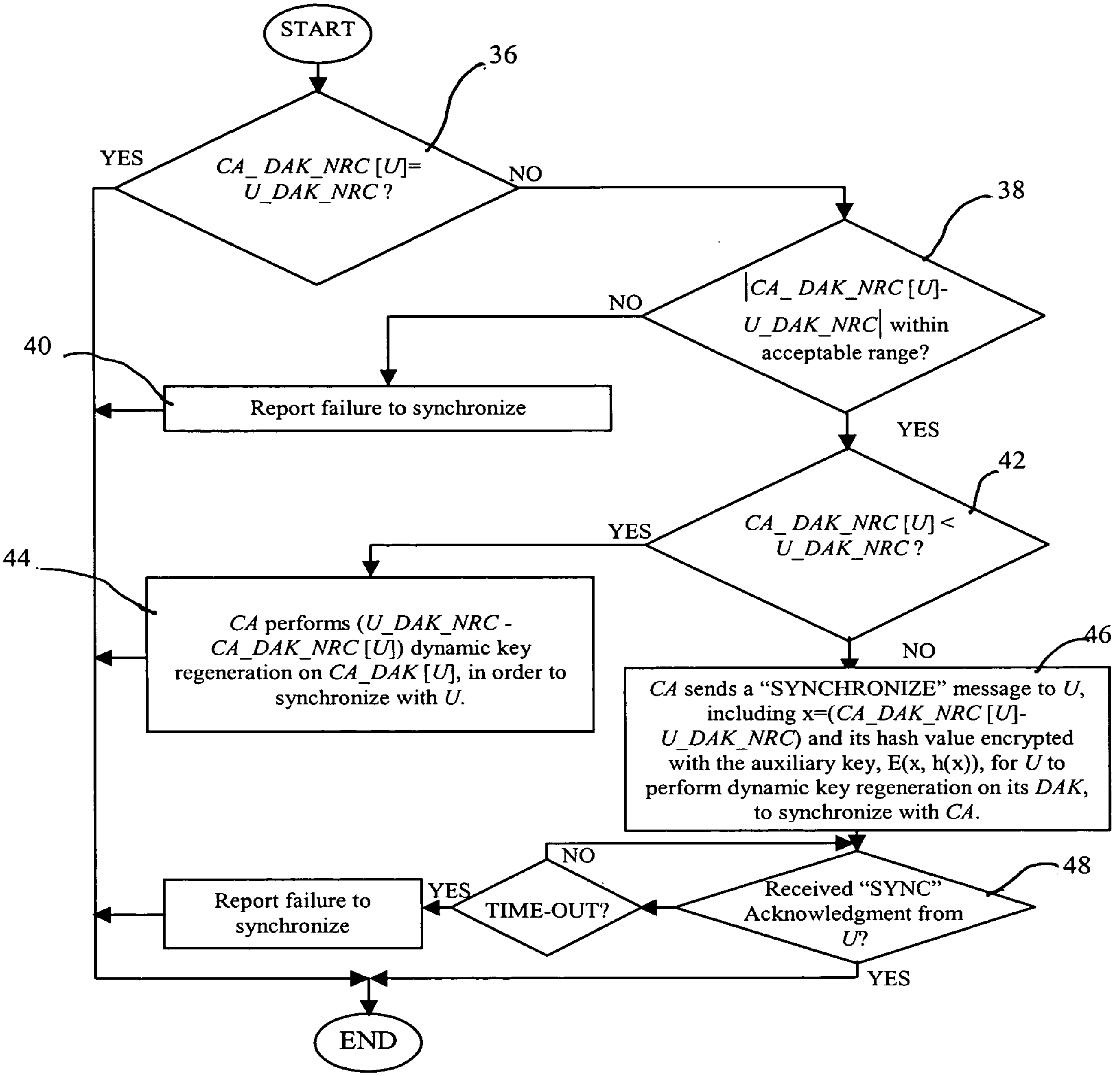
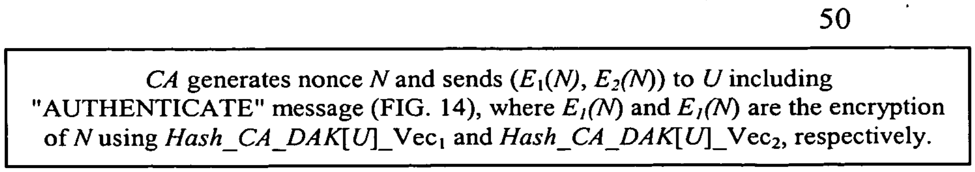

START
36
YES
CA_ DAK_NRC [*U*]=
U_DAK_NRC ?
NO
38
NO
|*CA_ DAK_NRC* [*U*]-
U_DAK_NRC| within
acceptable range?
40
Report failure to synchronize
YES
42
YES
CA_ DAK_NRC [*U*] <
U_DAK_NRC ?
44
CA performs (*U_DAK_NRC* - *CA_DAK_NRC* [*U*]) dynamic key regeneration on *CA_DAK* [*U*], in order to synchronize with *U*.
NO
46
CA sends a "SYNCHRONIZE" message to *U*, including x=(*CA_DAK_NRC* [*U*]- *U_DAK_NRC*) and its hash value encrypted with the auxiliary key, E(x, h(x)), for *U* to perform dynamic key regeneration on its *DAK*, to synchronize with *CA*.
NO
48
Report failure to synchronize
YES
TIME-OUT?
Received "SYNC" Acknowledgment from *U*?
YES
END

FIG. 7


50
CA generates nonce *N* and sends ($E_1(N)$, $E_2(N)$) to *U* including "AUTHENTICATE" message (FIG. 14), where $E_1(N)$ and $E_1(N)$ are the encryption of *N* using *Hash_CA_DAK*[*U*]_Vec$_1$ and *Hash_CA_DAK*[*U*]_Vec$_2$, respectively.

Hamdy Soliman

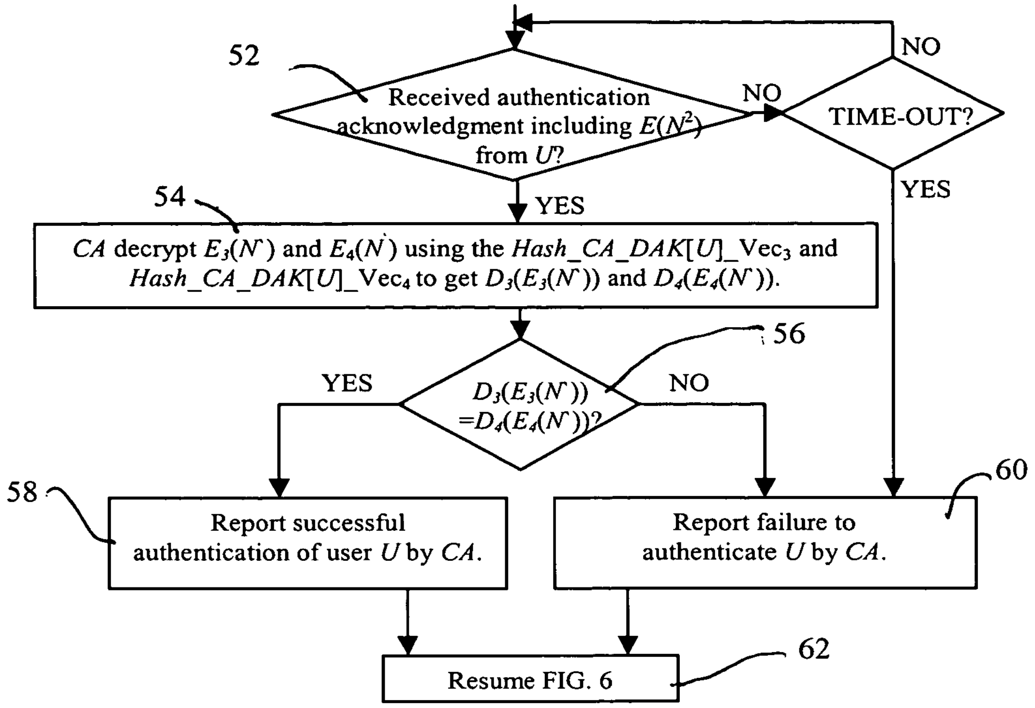

52
Received authentication acknowledgment including $E(N^2)$ from U?
NO
TIME-OUT?
NO
YES
YES
54
CA decrypt $E_3(N')$ and $E_4(N')$ using the $Hash\_CA\_DAK[U]\_Vec_3$ and $Hash\_CA\_DAK[U]\_Vec_4$ to get $D_3(E_3(N'))$ and $D_4(E_4(N'))$.
56
YES
$D_3(E_3(N'))$ $=D_4(E_4(N'))$?
NO
58
Report successful authentication of user U by CA.
60
Report failure to authenticate U by CA.
62
Resume FIG. 6

FIG. 8a

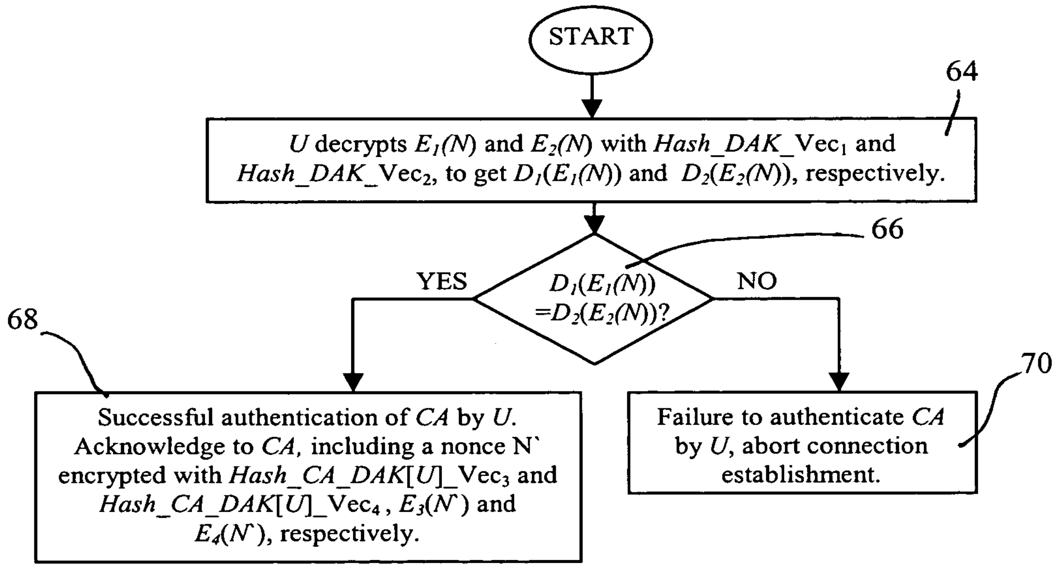

START
64
U decrypts $E_1(N)$ and $E_2(N)$ with $Hash\_DAK\_Vec_1$ and $Hash\_DAK\_Vec_2$, to get $D_1(E_1(N))$ and $D_2(E_2(N))$, respectively.
66
YES
$D_1(E_1(N))$ $=D_2(E_2(N))$?
NO
68
Successful authentication of CA by U. Acknowledge to CA, including a nonce N` encrypted with $Hash\_CA\_DAK[U]\_Vec_3$ and $Hash\_CA\_DAK[U]\_Vec_4$, $E_3(N')$ and $E_4(N')$, respectively.
70
Failure to authenticate CA by U, abort connection establishment.

FIG. 8b

Hamdy Soliman

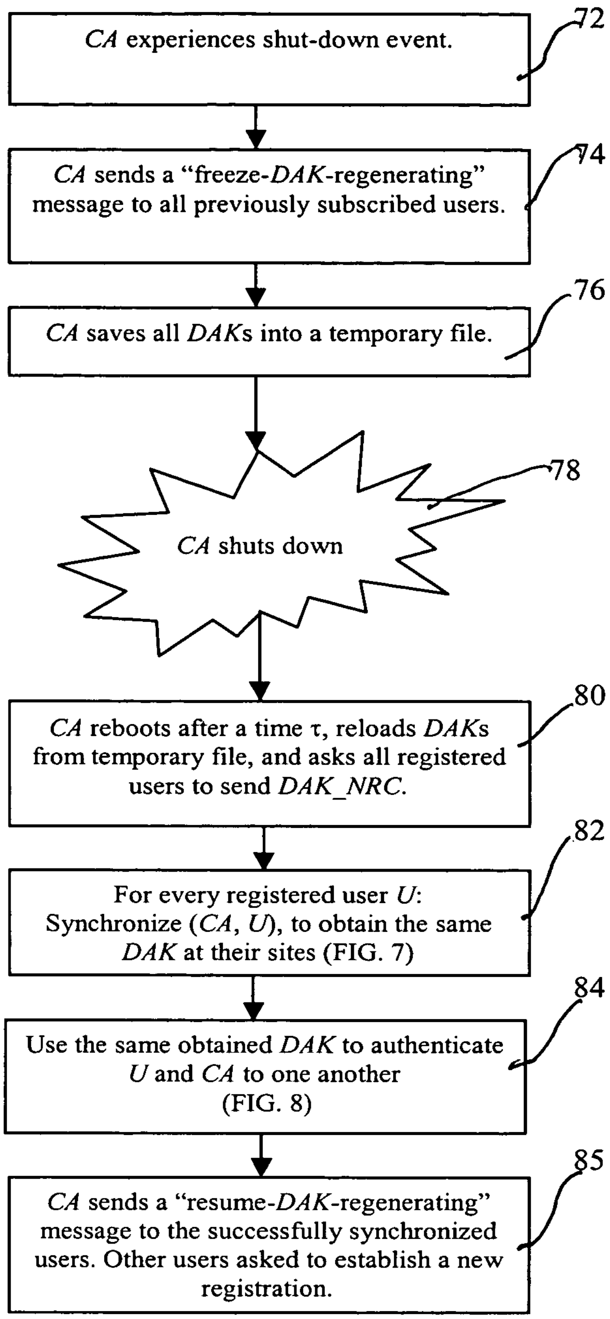

CA experiences shut-down event.
72
CA sends a "freeze-DAK-regenerating" message to all previously subscribed users.
74
CA saves all DAKs into a temporary file.
76
CA shuts down
78
CA reboots after a time τ, reloads DAKs from temporary file, and asks all registered users to send DAK_NRC.
80
For every registered user U: Synchronize (CA, U), to obtain the same DAK at their sites (FIG. 7)
82
Use the same obtained DAK to authenticate U and CA to one another (FIG. 8)
84
CA sends a "resume-DAK-regenerating" message to the successfully synchronized users. Other users asked to establish a new registration.
85

FIG. 9a

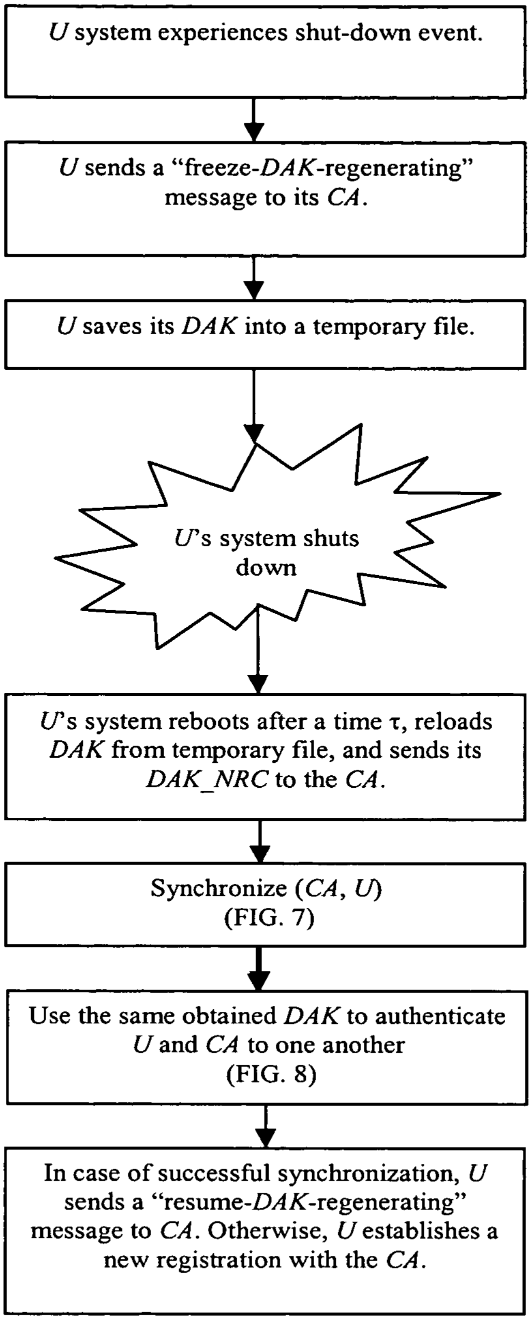

U system experiences shut-down event.
U sends a "freeze-DAK-regenerating" message to its CA.
U saves its DAK into a temporary file.
U's system shuts down
U's system reboots after a time τ, reloads DAK from temporary file, and sends its DAK_NRC to the CA.
Synchronize (CA, U) (FIG. 7)
Use the same obtained DAK to authenticate U and CA to one another (FIG. 8)
In case of successful synchronization, U sends a "resume-DAK-regenerating" message to CA. Otherwise, U establishes a new registration with the CA.

FIG. 9b

Handy Soliman

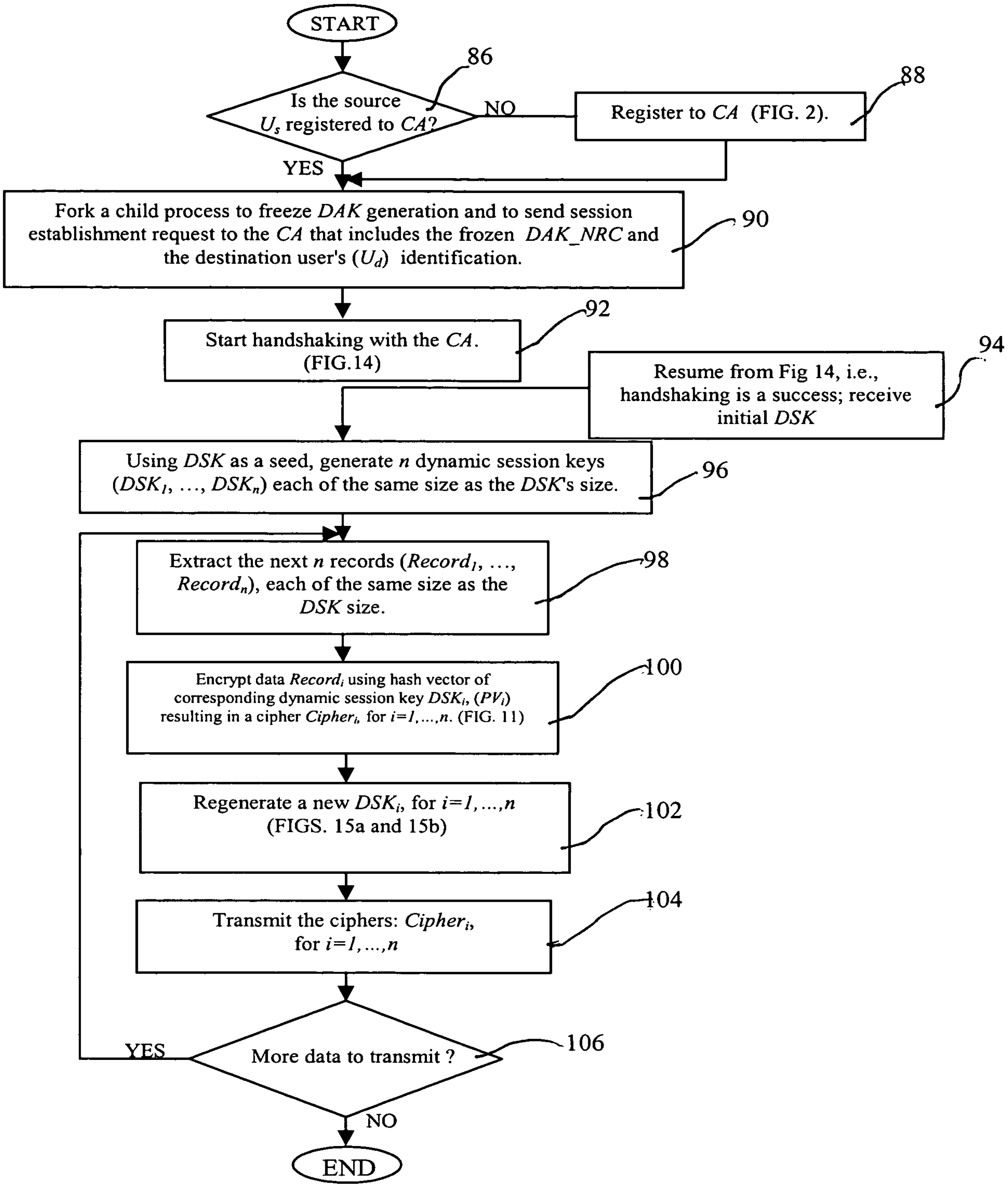

START
Is the source U_s registered to CA?
86
NO
Register to CA (FIG. 2).
88
YES
Fork a child process to freeze DAK generation and to send session establishment request to the CA that includes the frozen DAK_NRC and the destination user's (U_d) identification.
90
Start handshaking with the CA. (FIG.14)
92
Resume from Fig 14, i.e., handshaking is a success; receive initial DSK
94
Using DSK as a seed, generate n dynamic session keys (DSK_1, …, DSK_n) each of the same size as the DSK's size.
96
Extract the next n records ($Record_1$, …, $Record_n$), each of the same size as the DSK size.
98
Encrypt data $Record_i$ using hash vector of corresponding dynamic session key DSK_i, (PV_i) resulting in a cipher $Cipher_i$, for $i=1,\ldots,n$. (FIG. 11)
100
Regenerate a new DSK_i, for $i=1,\ldots,n$ (FIGS. 15a and 15b)
102
Transmit the ciphers: $Cipher_i$, for $i=1,\ldots,n$
104
YES
More data to transmit ?
106
NO
END

FIG. 10

Hamdy Soliman

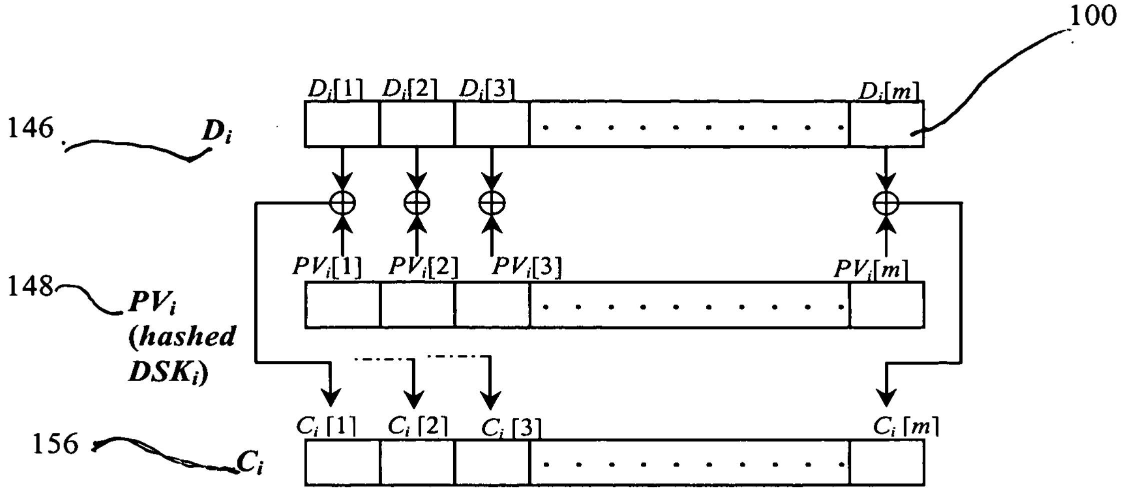


FIG. 11

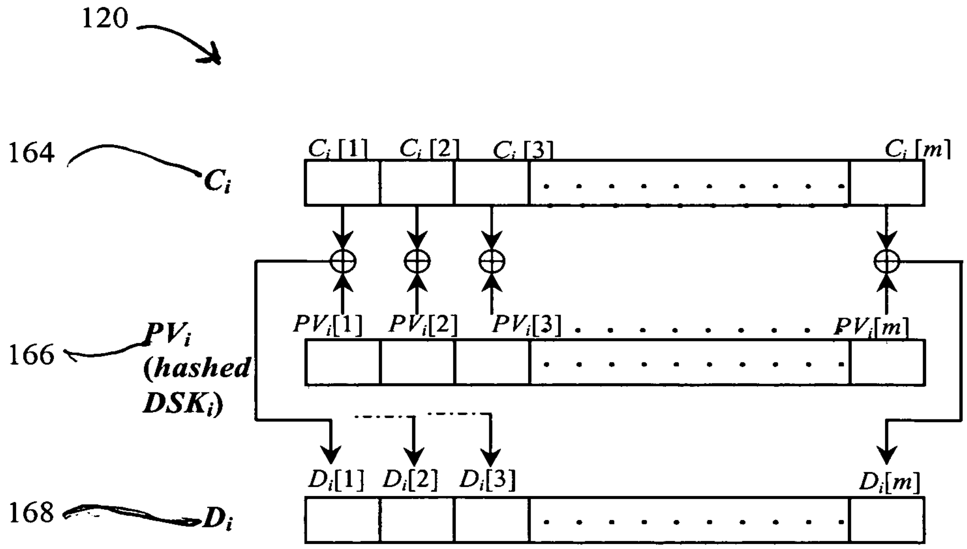


FIG. 13

Handy Soliman

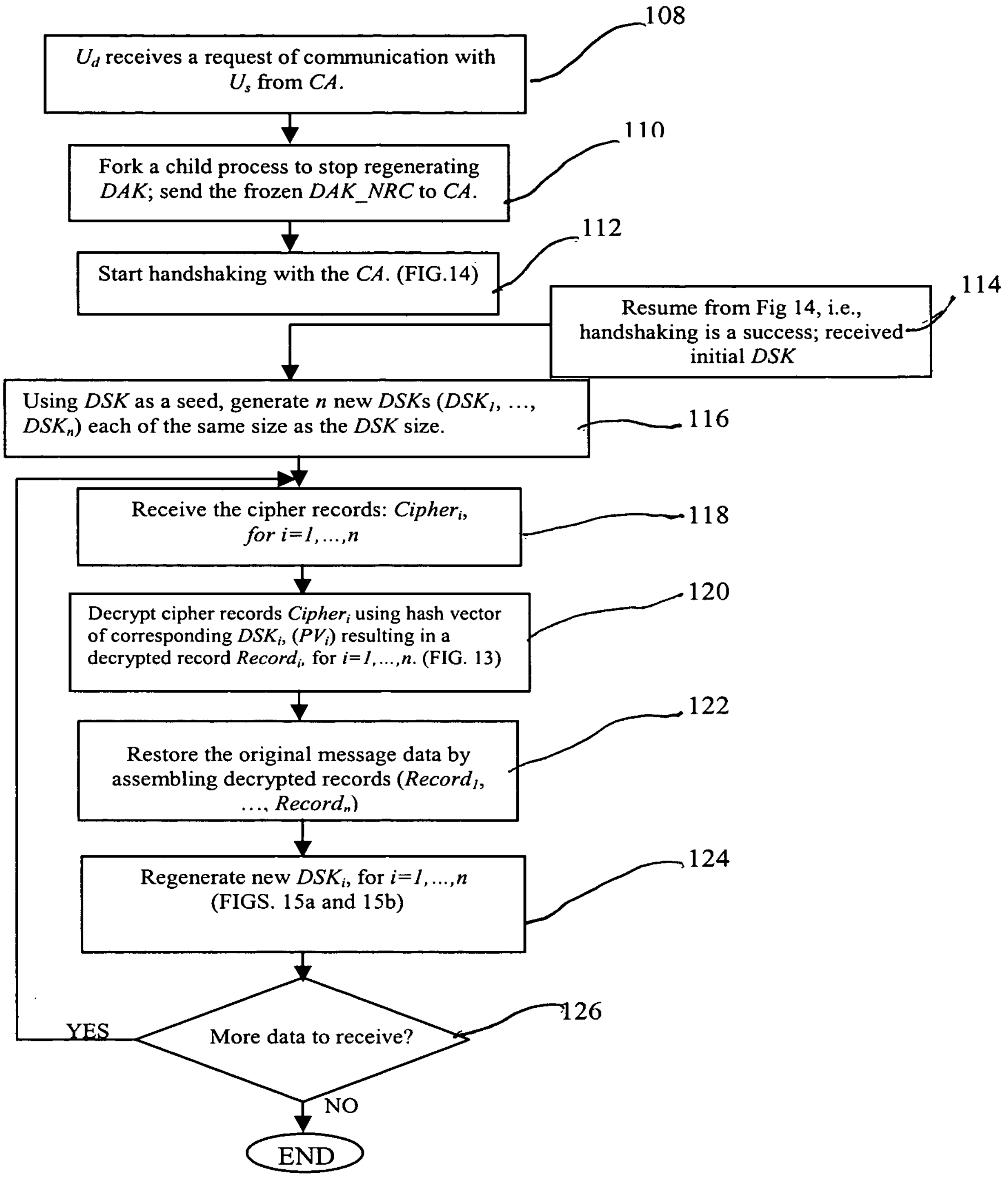

108
U_d receives a request of communication with U_s from *CA*.
110
Fork a child process to stop regenerating *DAK*; send the frozen *DAK_NRC* to *CA*.
112
Start handshaking with the *CA*. (FIG.14)
114
Resume from Fig 14, i.e., handshaking is a success; received initial *DSK*
Using *DSK* as a seed, generate *n* new *DSK*s (DSK_1, …, DSK_n) each of the same size as the *DSK* size.
116
Receive the cipher records: $Cipher_i$, *for i=1,…,n*
118
Decrypt cipher records $Cipher_i$ using hash vector of corresponding DSK_i, (PV_i) resulting in a decrypted record $Record_i$, for *i=1,…,n*. (FIG. 13)
120
Restore the original message data by assembling decrypted records ($Record_1$, …, $Record_n$)
122
Regenerate new DSK_i, for *i=1,…,n* (FIGS. 15a and 15b)
124
YES
More data to receive?
126
NO
END

FIG. 12

Handy Soliman

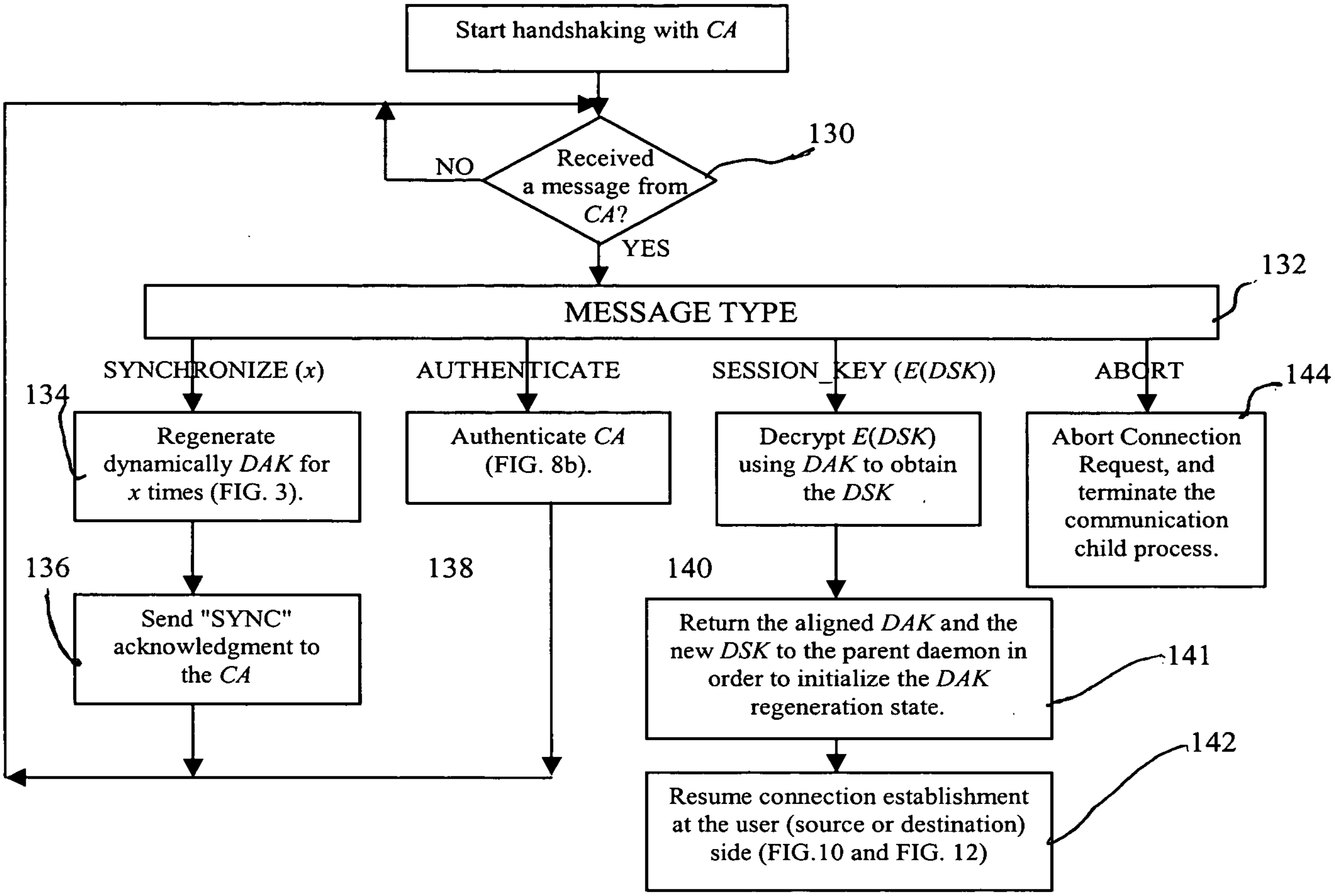

Start handshaking with *CA*
Received a message from *CA*?
130
NO
YES
MESSAGE TYPE
132
SYNCHRONIZE (*x*)
AUTHENTICATE
SESSION_KEY (*E*(*DSK*))
ABORT
134
Regenerate dynamically *DAK* for *x* times (FIG. 3).
Authenticate *CA* (FIG. 8b).
Decrypt *E*(*DSK*) using *DAK* to obtain the *DSK*
144
Abort Connection Request, and terminate the communication child process.
136
138
140
Send "SYNC" acknowledgment to the *CA*
Return the aligned *DAK* and the new *DSK* to the parent daemon in order to initialize the *DAK* regeneration state.
141
142
Resume connection establishment at the user (source or destination) side (FIG.10 and FIG. 12)

FIG. 14

Hamdy Soliman

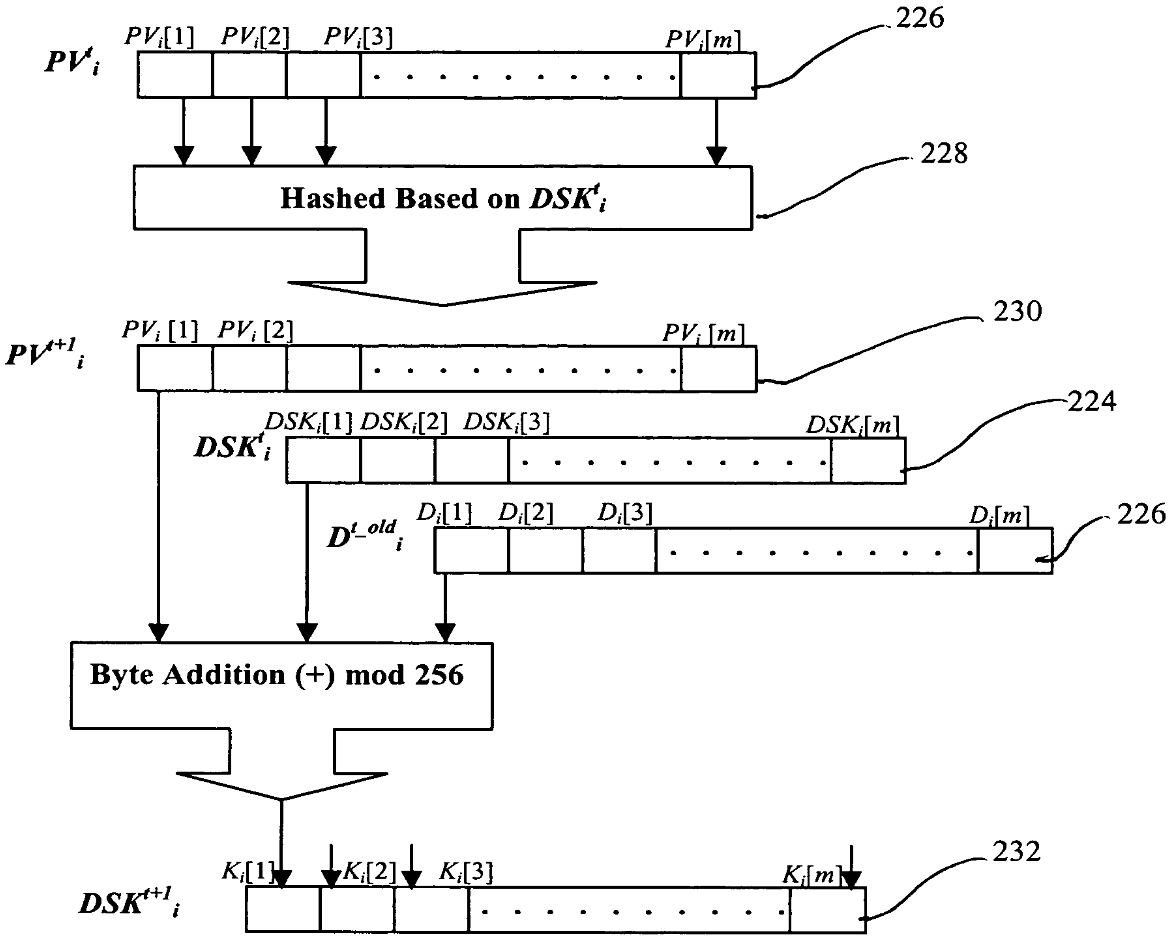
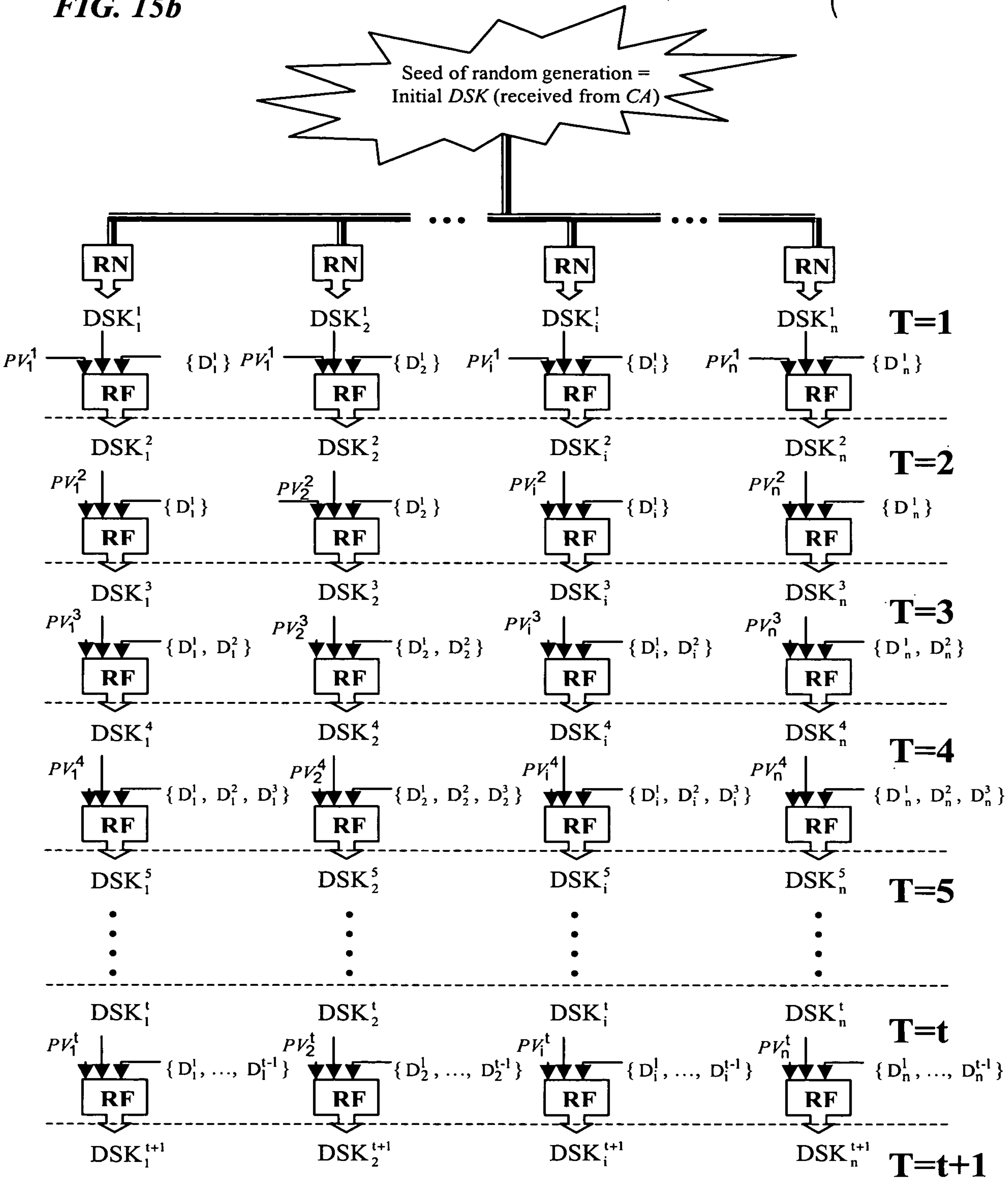

PV^t_i
$PV_i[1]$ $PV_i[2]$ $PV_i[3]$ $PV_i[m]$
226
Hashed Based on DSK^t_i
228
PV^{t+1}_i
$PV_i\,[1]$ $PV_i\,[2]$ $PV_i\,[m]$
230
DSK^t_i
$DSK_i[1]$ $DSK_i[2]$ $DSK_i[3]$ $DSK_i[m]$
224
$D^{t\_old}_i$
$D_i[1]$ $D_i[2]$ $D_i[3]$ $D_i[m]$
226
Byte Addition (+) mod 256
DSK^{t+1}_i
$K_i[1]$ $K_i[2]$ $K_i[3]$ $K_i[m]$
232

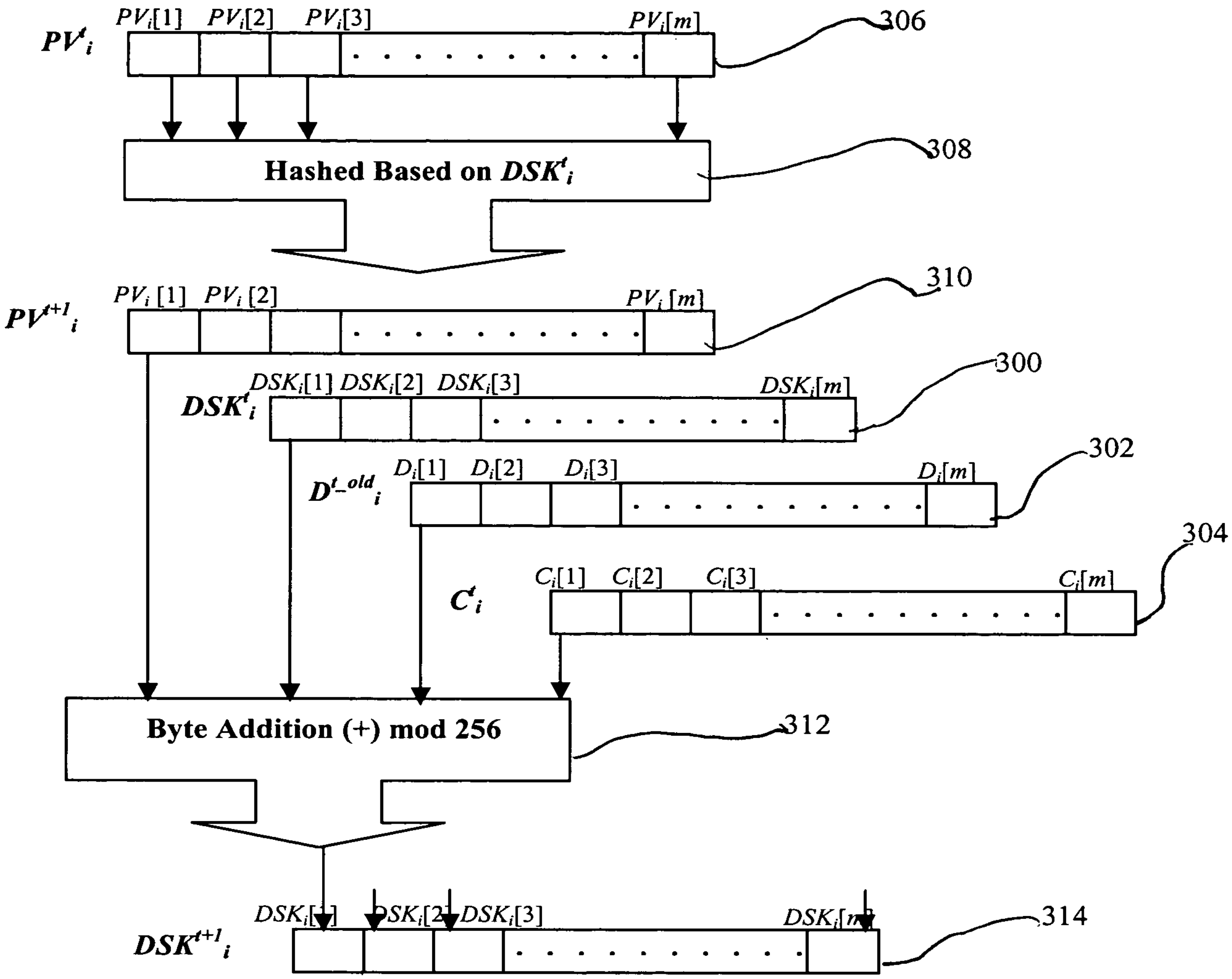
FIG. 15a

16/19 Hamdy Soliman

FIG. 15b


Seed of random generation = Initial *DSK* (received from *CA*)
RN
RF
DSK_1^1 DSK_2^1 DSK_i^1 DSK_n^1
PV_1^1 $\{D_1^1\}$ PV_1^1 $\{D_2^1\}$ PV_i^1 $\{D_i^1\}$ PV_n^1 $\{D_n^1\}$
DSK_1^2 DSK_2^2 DSK_i^2 DSK_n^2
PV_1^2 $\{D_1^1\}$ PV_2^2 $\{D_2^1\}$ PV_i^2 $\{D_i^1\}$ PV_n^2 $\{D_n^1\}$
DSK_1^3 DSK_2^3 DSK_i^3 DSK_n^3
PV_1^3 $\{D_1^1, D_1^2\}$ PV_2^3 $\{D_2^1, D_2^2\}$ PV_i^3 $\{D_i^1, D_i^2\}$ PV_n^3 $\{D_n^1, D_n^2\}$
DSK_1^4 DSK_2^4 DSK_i^4 DSK_n^4
PV_1^4 $\{D_1^1, D_1^2, D_1^3\}$ PV_2^4 $\{D_2^1, D_2^2, D_2^3\}$ PV_i^4 $\{D_i^1, D_i^2, D_i^3\}$ PV_n^4 $\{D_n^1, D_n^2, D_n^3\}$
DSK_1^5 DSK_2^5 DSK_i^5 DSK_n^5
DSK_1^t DSK_2^t DSK_i^t DSK_n^t
PV_1^t $\{D_1^1, \ldots, D_1^{t-1}\}$ PV_2^t $\{D_2^1, \ldots, D_2^{t-1}\}$ PV_i^t $\{D_i^1, \ldots, D_i^{t-1}\}$ PV_n^t $\{D_n^1, \ldots, D_n^{t-1}\}$
DSK_1^{t+1} DSK_2^{t+1} DSK_i^{t+1} DSK_n^{t+1}
T=1
T=2
T=3
T=4
T=5
T=t
T=t+1

Hamdy Soliman



PV^t_i
$PV_i[1]$ $PV_i[2]$ $PV_i[3]$ $PV_i[m]$
306
Hashed Based on DSK^t_i
308
PV^{t+1}_i
PV_i [1] PV_i [2] PV_i [m]
310
DSK^t_i
$DSK_i[1]$ $DSK_i[2]$ $DSK_i[3]$ $DSK_i[m]$
300
$D^{t\_old}_i$
$D_i[1]$ $D_i[2]$ $D_i[3]$ $D_i[m]$
302
C^t_i
$C_i[1]$ $C_i[2]$ $C_i[3]$ $C_i[m]$
304
Byte Addition (+) mod 256
312
DSK^{t+1}_i
$DSK_i[1]$ $DSK_i[2]$ $DSK_i[3]$ $DSK_i[m]$
314

FIG. 16

Hamdy Soliman

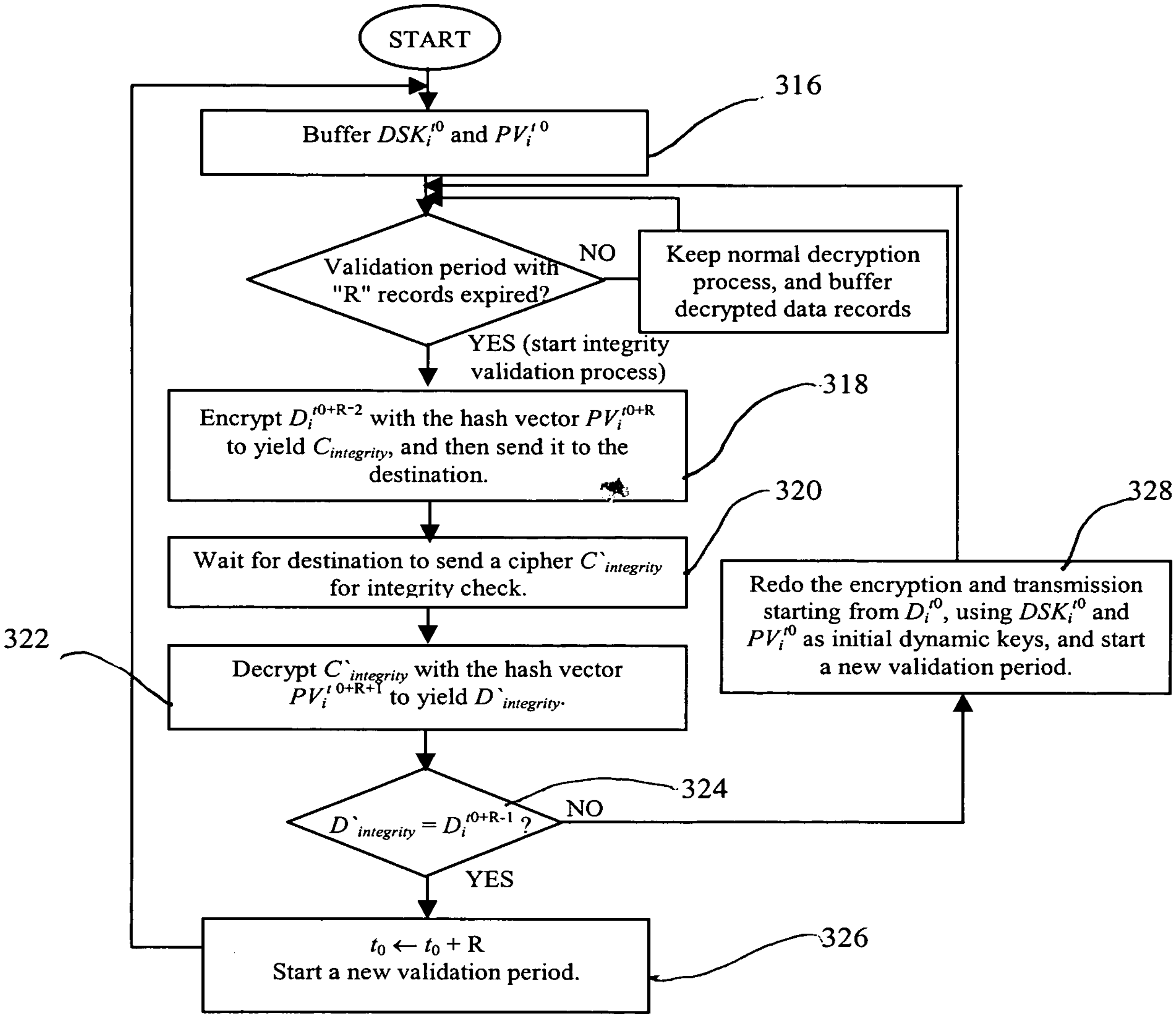

START
Buffer DSK_i^{t0} and PV_i^{t0}
316
Validation period with "R" records expired?
NO
Keep normal decryption process, and buffer decrypted data records
YES (start integrity validation process)
Encrypt D_i^{t0+R-2} with the hash vector PV_i^{t0+R} to yield $C_{integrity}$, and then send it to the destination.
318
Wait for destination to send a cipher $C`_{integrity}$ for integrity check.
320
328
Redo the encryption and transmission starting from D_i^{t0}, using DSK_i^{t0} and PV_i^{t0} as initial dynamic keys, and start a new validation period.
322
Decrypt $C`_{integrity}$ with the hash vector $PV_i^{t0+R+1}$ to yield $D`_{integrity}$.
324
$D`_{integrity} = D_i^{t0+R-1}$?
NO
YES
$t_0 \leftarrow t_0 + R$
Start a new validation period.
326

FIG. 17a

Hamdy Soliman

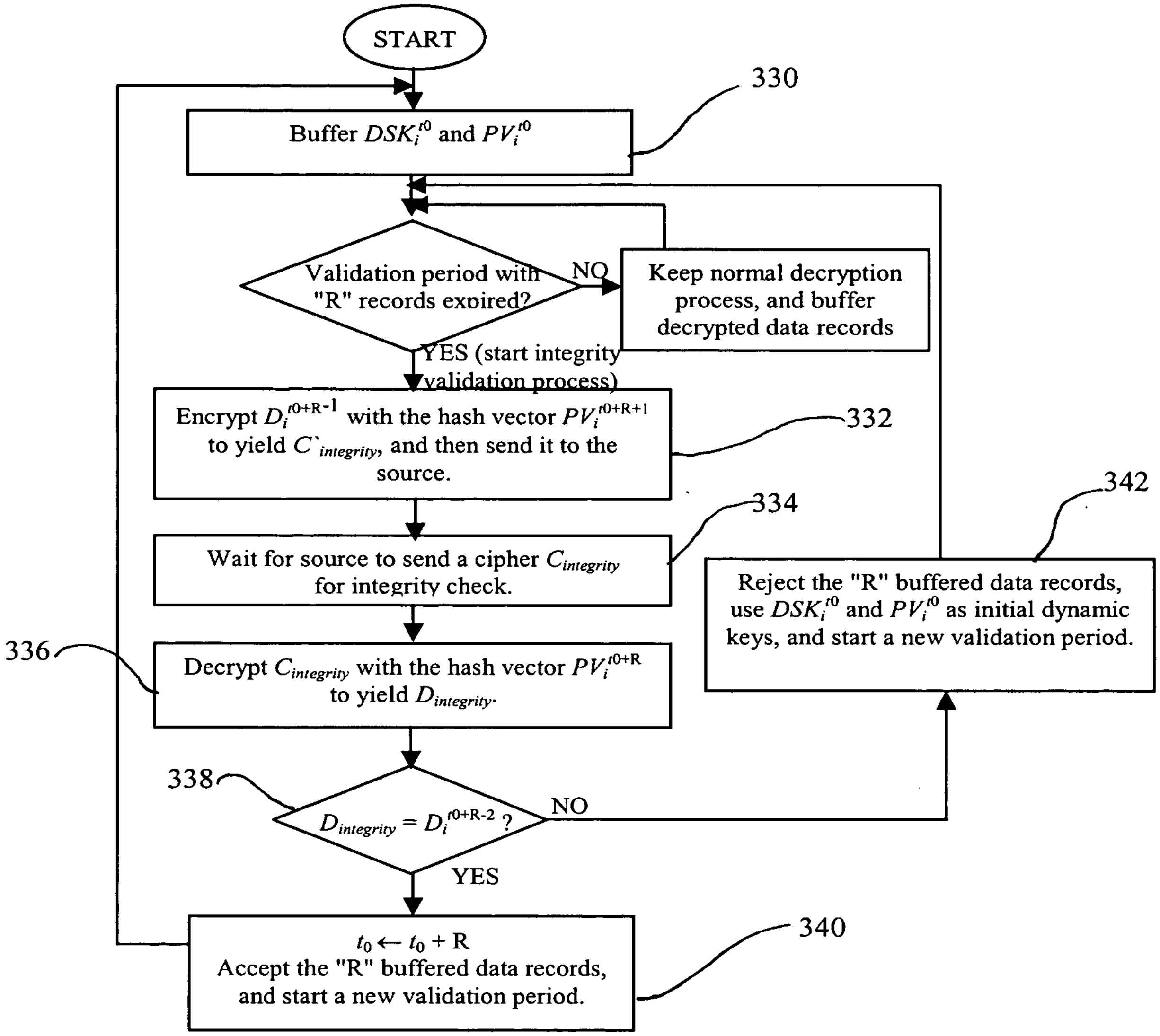

START
330
Buffer DSK_i^{t0} and PV_i^{t0}
Validation period with "R" records expired?
NO
Keep normal decryption process, and buffer decrypted data records
YES (start integrity validation process)
Encrypt D_i^{t0+R-1} with the hash vector PV_i^{t0+R+1} to yield $C`_{integrity}$, and then send it to the source.
332
334
Wait for source to send a cipher $C_{integrity}$ for integrity check.
342
Reject the "R" buffered data records, use DSK_i^{t0} and PV_i^{t0} as initial dynamic keys, and start a new validation period.
336
Decrypt $C_{integrity}$ with the hash vector PV_i^{t0+R} to yield $D_{integrity}$.
338
$D_{integrity} = D_i^{t0+R-2}$?
NO
YES
$t_0 \leftarrow t_0 + R$
Accept the "R" buffered data records, and start a new validation period.
340

FIG. 17b